Java同步线程模型分析与改进

陈文宇，陈 福，余盛季，孙世新

(电子科技大学计算机科学与工程学院 成都 610054)

【摘要】目前普遍采用急救包(Band-Aid)类库的方式解决Java 线程模型存在的同步问题，但类库中的代码很难或无法实现优化。该文针对Java同步线程模型的缺陷，扩展synchronised关键字语法，使它支持多个参数和能接受一个超时说明；重新定义wait()使它返回一个boolean变量来解决超时检测问题；通过扩展语法方法解决了同步问题，以确保使用Java线程所开发的程序的稳定、可靠和可优化。

关 键 词 双检锁； Java内存模型； Java线程模型； 同步

中图分类号 TP312.1； TP309 **文献标识码** A doi:10.3969/j.issn.1001-0548.2010.03.023

Analysis and Improvement for Synchronous Thread Model Based on Java

CHEN Wen-yu, CHEN Fu, YU Sheng-ji, and SUN Shi-xin

(School of Computer Science and Engineering, University of Electronic Science & Technology of China Chengdu 610054)

Abstract Java supports threading at language level. In order to synchronize, Java provides keyword “synchronized” and mechanisms like wait() for Object, however, the potential imperfectness of the mechanisms may trigger unpredictable results. Currently, “Band Aid” class library is often used to solve the synchronize problem of Java thread model, but it is hard to optimize the class library. The paper discusses drawbacks of Java threading mechanism, extends the grammar of synchronized keyword for accepting multiple parameters and a timeout callback; redefines wait() method for returning a boolean variable to solve the timeout detection issue. The synchronize issue is solved by using syntax extension methods to guarantee the stability, reliability and optimized attributes of the programs developed by Java threads.

Key words double-checked locking; Java memory model; Java thread model; synchronization

尽管Java在语言级别就支持多线程编程，但是它对线程的语法和类包的支持太少，只能适用于极小型的应用环境，完全不能满足实际复杂程序的要求，而且不完全面向对象。大多数关于Java线程编程的书籍都指出了Java线程模型的缺陷，并提供了解决这些问题的急救包(band-aid)类库。由于编译器和Java虚拟机(JVM)能一同优化程序代码，而优化对于类库中的代码是很难或无法实现的，本文提出扩展语法而不是提供类库的方法，改进Java的同步线程模型，可产生更高效和可靠的代码。

1 Java内存管理

1.1 Java内存模型

Java内存模型(Java memory model)中，Java作为平台无关性语言，根据Java语言规范(JLS)，它定义一个统一的内存管理模型，具体由各个虚拟机来实现。Java平台最大的特点就是不用程序员显示地释放与回收内存。Java内存模型的设计在方便开发人员的同时，也增加了虚拟机实现的复杂程度，虚拟机在做代码优化时还有可能导致一些问题的出现。Java内存模型定义了线程访问内存的规则与方式。对于开发人员，只需要在需要同步的代码段给出同步关键字，虚拟机便会在内部帮其实现比较复杂的诸如对象的加锁与解锁等操作。

JMM将线程能访问的内存划分为主内存(main memory)与工作内存(working memory)。在主内存中，存放程序中所有的类实例、静态数据等变量，所有线程都可以对主存中的数据进行并发访问。而每一个线程都有一个私有的其他线程不能访问的工作内存，其中存放的是该线程从主内存中拷贝过来的变量以及访问方法所取得的局部变量，每个线程对变量的操作都是以先从主内存将其拷贝到工作内

收稿日期：2008－10－24； 修回日期：2009－09－13

基金项目：国家863计划(2007AA01Z131)

作者简介：陈文宇(1968－)，男，博士，副教授，主要从事编译技术、模式识别、形式语言与自动机等方面的研究.

存再对其进行操作的方式进行，各线程之间想要读取或者修改对方所访问的变量均需通过主内存。

当需要读取数据时就从主内存拷贝一份需要的变量到工作内存。当需要修改某变量值时，首先从主内存复制变量到工作内存，修改其值后将其拷贝回主内存。JLS中对线程操作主存定义了load、save、read、write、assign和use6种行为。这些行为是不可分解的原子操作，在使用上相互依赖，有具体的规范约束[1-2]。

如果某个线程访问一个标识为synchronized的方法，并对相应变量做操作，那么根据Java语言规范，JVM的执行步骤如下：

(1) 取得该对象锁(普通方法的锁为this对象，静态方法则为该类的class对象)并将其锁住(lock)。

(2) 将需要的数据从主内存拷贝到自己的工作内存(read and load)。

(3) 根据程序流程读取或者修改相应变量值(use and assign)。

(4) 将自己工作内存中修改了值的变量拷贝回主内存(store and write)。

(5) 释放对象锁(unlock)。

由此可见，synchronized的一个重要作用是保证它所锁住的代码区的访问的唯一性，即任何时刻只允许唯一的一个线程单独访问同步区。

在不使用synchronized关键字的情况下，JVM就不能保证步骤(2)和步骤(4)会按文中所描述的次序立即执行。根据JLS规定，线程的工作内存与主内存之间的数据交换是松耦合的，刷新工作内存或者更新主内存内容的时机，由具体的虚拟机实现决定。

在多个线程并发访问某个代码段时，有可能出现某个线程在自己的工作内存中修改了变量值，但是没有及时刷新到主内存，导致其他线程在并发访问中读取的是该变量修改之前的值。这种对数据修改的不可见性会导致程序运行结果的不可靠性和不一致性。

1.2 DCL(double-checked locking)失效

LazyLoad技巧是指类中的某些成员变量在类文件加载进JVM的初期不会为这些变量赋值，只有在程序中的某个地方要首次使用这些变量时才会把它们的值加载进来。例如下面的代码1：

```
class Foo {
    private Helper helper=null;
    public Helper getHelper() {
        if (helper==null)
            res=new Helper();
        return helper;
    }
}
```

该代码在单线程环境下可行，但是在多线程环境下执行该代码就可能出现一些不可预见的问题。比如执行至if(helper==null)时，由于多个线程可能同时访问该代码，符合判断条件进入随后的代码块，就有可能出现helper被多次初始化的问题。为了避免这样的Race Condition，需要用synchronized对各种方法进行同步。由于synchronized标识的方法可能会因为对象锁的获取与释放使得执行效率降低，所以应该只在首次调用该方法对变量helper进行初始化时才进行同步。初始化完成以后，helper变量不再为null，就不会再进入同步块生成新的实例。重构代码1为代码2：

```
class Foo {
    private Helper helper=null;
    public Helper getHelper() {
        if (helper==null) {
            synchronized (this) {
                if (helper==null)
                    res=new Resource();
            }
        }
        return res;
    }
}
```

这种重构技巧就是DCL(double-checked locking)[3-5]。但是代码2是不可靠的。

造成重构后的代码2不可靠的重要原因是相关的编译器可能在编译程序生成机器代码时的次序做了调整。只要在单个线程情况下执行结果是正确的，就可以认为编译器是合法的。JLS在某些方面的规定比较自由，就是为了让JVM有更多进行代码优化以提高执行效率的余地。而现代的CPU越来越多地采用RISC，CPU内部的流水线超过了以往旧式CPU的5～6步的指令流水线，达到了20步以上的流水线。超标量的CPU在一个时钟周期内可以执行一条以上的指令。为了适应这类CPU的快速指令处理流水线，多数编译器采取了对编译生成的机器指令进行优化调整的方式，以便在程序处理时保证程序的执行与快速的指令处理流水线相适应，这种做法使程序执行效率提升，但也导致了采用DCL方式重构代码的

不可靠性。

为了进一步证明这个问题，引用《DCL Broken Declaration》[4]的例子。

objects[i].reference=new Object()：

经过Symantec JIT编译器编译过以后，最终生成汇编码在机器中执行：

0206106A mov eax, 0F97E78h

0206106F call 01F6B210

为Object申请内存空间。返回值放在eax中

02061074 mov dword ptr [ebp], eax

ebp中是objects[i].reference的地址，将返回的空间地址放入其中，此时Object尚未初始化。

02061077 mov ecx, dword ptr [eax]

引用eax所指向的内容，获得新对象的起始地址。

内联的构造函数为：

02061079 mov dword ptr [ecx], 100h

0206107F mov dword ptr [ecx+4], 200h

02061086 mov dword ptr [ecx+8], 400h

0206108D mov dword ptr [ecx+0Ch], 0F84030h

虽然Object构造函数尚未调用，但已经能够通过objects[i].reference获得Object对象实例引用。

如果把代码放到多线程环境下运行，某线程在执行到该行代码时，JVM或者操作系统进行一次线程切换，其他线程显然会发现对象已经不为空，导致Lazy Load的判空语句不成立。线程认为对象已经建立成功，随之可能会使用对象的成员变量或者调用该对象实例的方法，最终导致不可预测的错误。

导致DCL失效的原因是在真正的多CPU环境下，每个CPU都使用自己的Cache和寄存器，但却共享系统内存。所以CPU在执行指令时可能会对指令做出一定的调整，以便能将分配到各个CPU的指令执行结果与共享的系统内存同步[6-7]。可以把main memory看作系统的物理内存，把thread working memory认为是CPU内部的Cache和寄存器，没有synchronized的保护，Cache和寄存器的内容就不会及时与主内存的内容同步，从而导致一条线程无法看到另一条线程对一些变量的改动。

1.3 wait()超时的语意模糊性

Java在Object类中提供wait()、notify()和notifyAll()，使得所有的类都隐式地继续这些方法。为了提供对程序健壮性方面的考虑，在Java中提供了对wait方法超时语意的支持。但是Java在对wait方法超时语意的支持方面存在模糊性，即在调用具有超时语意的wait方法返回时，无法区分是由于notify()/notifyAll()的通知还是由于超时触发的。

例如，在开发一个应用服务器时，实现能够高效地处理来自多个客户端的并发请求的功能。为了简化同步控制，分离并发逻辑和业务逻辑，采用Active Object模式。开发设计中有一个核心部件ActiveQueue用于存放客户的请求。为了能够做到运用服务器的负载控制，需对ActiveQueue的大小进行限制。假如当前的客户请求数量已经达到该限制，就让后继的请求等待，具体的代码实现片断如下(为了简洁起见，省略了其他无关的代码)。

```
public synchronised enqueue (clientQuest cr) thwors
    InterruptedException
{
    while (isFull())//ActiveQueue的大小达到上限
    {
        wait();
    }
    //把用户请求添加到处理队列中
    notifyAll();
}
```

在测试中发现了两个问题：(1) 当并发请求的客户端很多时，会造成某些客户端等待的时间过长，对于客户端的使用者非常不友好；(2) 由于系统中应用服务器的其他方面的异常同样会造成客户端请求的永久等待，如应用服务器在处理完客户端请求后，由于异常没有正确地调用相应的notifyAll方法。为了改善程序的用户友好性及健壮性，应采用带有超时语意的wait方法。该方法的原型声明如下：

```
public synchronised enqueue (clientQuest cr) thwors
    InterruptedException
{
    while (isFull())//ActiveQueue的大小达到上限
    {
        wait(timeOut)；//语意模糊体现于此
    //当wait()返回时，无法区分是由于notifyAll()的
    //通知还是超时触发的，因此无法做出适当处理
    }
    //把用户请求添加到处理队列中
    notifyAll()；
}
```

简单地使用一个具有超时语意的wait方法是不可行的，原因就在于wait方法超时语意的模糊性。

2 改进方案

2.1 扩展synchronised关键字语法

扩展synchronized 的语法，使其支持多个参数和能接受一个超时说明(在括弧中指定)[5]。

synchronized(*x*&&*y*&&*z*)，获得*x*、*y*和*z*对象的锁。

synchronized(*x*||*y*||*z*)，获得*x*或*y*或*z*对象的锁。

synchronized((*x*&&*y*)||*z*)，获得*x*、*y*或*z*对象的锁。

synchronized(...)[1 000]，设置1 s超时以获得一个锁。

synchronized[1 000]*f*(){…}在进入*f*()函数时获得this的锁，但可有1 s的超时。

TimeoutException 是 RuntimeException 的派生类，它在等待超时后即被抛出。

超时是需要的，但还不足以使代码强壮，还需要具备从外部中止请求锁等待的能力。所以，当向一个等待锁的线程传送一个interrupt()方法后，该方法应抛出一个SynchronizationException对象，并中断等待的线程。这个异常是RuntimeException的一个派生类，不需要特别处理。

对synchronized语法推荐的这些更改方法的主要问题是，它们需要在二进制代码级上修改。目前这些代码使用进入监控(enter-monitor)和退出监控(exit-monitor)指令实现synchronized。但这些指令没有参数，所以需要扩展二进制代码的定义以支持多个锁定请求。

另一个可解决的问题是最常见的死锁情况，在该死锁情况下，两个线程都在等待对方完成某个操作[8-9]。考虑下面一个例子。

设想一个线程调用a()，但在获得lock 1之后、获得lock 2之前被剥夺运行权。第二个线程进入运行，调用b()，获得lock 2，但是由于第一个线程占用lock 1，所以它无法获得lock 1，随后处于等待状态。第一个线程被唤醒，它试图获得lock 2，但是由于被第二个线程占据，所以无法获得。此时出现死锁。

Synchronize-On-Multiple-Objects的语法可解决死锁问题。编译器(或虚拟机)会重新排列请求锁的顺序，使lock 1总是被首先获得，消除了死锁[10]。但是，该方法对多线程不一定总能成功，所以需要提供一些方法自动打破死锁。一个简单的办法就是在等待第二个锁时释放已获得的锁，即如果等待锁的每个程序使用不同的超时值，可打破死锁，其中一个线程就可运行。

2.2 改进wait()

超时检测问题可以通过重新定义wait()使它返回一个boolean变量(而不是void)来解决。一个为true的返回值表示一个正常返回，而false表示超时返回。

基于状态的条件变量的概念是很重要的。如果该变量被设置成false状态，那么等待的线程将要被阻断，直到该变量进入true状态。任何等待true的条件变量的等待线程会被自动释放，在这种情况下，wait()调用不会发生阻断。

嵌套监控锁定问题非常麻烦，暂时没有简单的解决办法。嵌套监控锁定是一种死锁形式，当某个锁的占有线程在挂起其自身之前不释放锁时，会发生嵌套监控封锁。考虑下面的例子。

在get()和put()操作中涉及两个锁：一个在Stack对象上，另一个在LinkedList对象上。考虑当一个线程试图调用一个空栈的pop()操作时的情况。该线程获得这两个锁，然后调用wait()释放Stack对象上的锁，但是没有释放在LinkedList上的锁。如果此时第二个线程试图向堆栈中压入一个对象，它会在synchronized(list)语句上永远挂起，而且永远不会被允许压入一个对象。由于第一个线程等待的是一个非空栈，就会发生死锁。这就是说，第一个线程永远无法从wait()返回，因为它占据着锁，而导致第二个线程永远无法运行到notify()语句。

一个可行的方案是在wait()中按照反顺序释放当前线程获取的所有锁，当等待条件满足后，重新按原始获取顺序取得它们。

3 结　语

本文针对Java同步线程机制的缺陷所引起的一些问题进行探讨，并给出了一些解决方案。改进语法而不是以第三方补丁包的形式对Java类库进行完善，将能产生更高效的代码。这是因为编译器和Java虚拟机能一同优化程序代码，而这些优化对于类库中的代码是很难或无法实现的。

本文的研究工作得到电子科技大学青年基金(L08010601JX0754)的资助，在此表示感谢。

参 考 文 献

[1] GOSLING J, JOY B, STEELE G, et al. Java language specification[M]. 3rd. ed. New York: Addison-Wesley Professional, 2005.

[2] GOSLING J, JOY B, STEELE G, et al. Java code conventions[M]. 3rd ed. New York: Addison-Wesley Professional, 2004.

[3] HORSTMANN C S, CORNELL G. Java2核心技术(卷2): 高级特性(第7版)[M]. 陈昊鹏, 王　浩, 姚建平, 等译.

北京: 机械工业出版社, 2006.
HORSTMANN C S, CORNELL G. Core of Java 2 (Volume 2): Advanced features[M]. 7th ed. Translated by CHEN Hao-peng, WANG Hao, YAO Jian-ping, et al. Beijing: China Machine Press. 2006.
[4] ALLEN H. Taming Java threads[M]. New York: [s.n.], 2000.
[5] DARBY C, GRIFFIN J, PASCAL de HAAN, 等. Java网络编程指南[M]. 邱仲潘译. 北京: 电子工业出版社, 2002.
DARBY C, GRIFFIN J, PASCAL de HAAN, et al. Guide of Java networking[M]. Translated by QIU Zhong-pan. Beijing: Publishing House of Electronic Industry, 2002.
[6]柴平洵, 龚向阳, 程时端. 分布式入侵检测技术的研究[J]. 北京邮电大学学报, 2002, 25(2): 58-62.
CHAI Ping-xuan, GONG Xiang-yang, CHENG Shi-duan. Research on distributed intrusion detection[J]. Journal of Beijing University of Posts and Telecommunications, 2002, 25(2): 58-62
[7] 喻志虎, 邹 华, 杨放春. Parlay应用服务器的软件容错研究与设计[J]. 北京邮电大学学报, 2004, 27(增刊): 12-15
YU Zhi-hu, ZOU Hua, YANG Fang-chun. The study and design on software fault-tolerance for parlay applica ti on server[J]. Journal of Beijing University of Posts and Telecommunications, 2004, 27(An Extra Edition): 12-15.
[8] VENNERS B. Inside the Java virtual machine[M]. 2nd ed. New York: McGraw-Hill Companies, 2003.
[9] GAY S H, CORNELL G. Core Java 2 volume II advanced features[M]. 6th ed. New York: Prentice Hall PTR, 2006.
[10] ECKEL B. Thinking in Java[M]. 3rd ed. New York: Prentice Hall PTR, 2005.

编 辑 黄 莘

(上接第406页)

[10] 才 辉, 张光新, 张 浩, 等. 一种新的基于多信息测度融合的边缘检测方法[J]. 浙江大学学报(工学版), 2008, 42(10): 1671- 1675.
CAI Hui, ZHANG Guang-xin, ZHANG Hao, et al. Novel edge detection method based on multiple information measures fusion[J]. 2008, 42(10): 1671-1675.
[11] 傅茂洺. 基于形态灰度边缘检测算法的一种改进[J]. 电子科技大学学报, 2005, 34(2): 206-209.
FU Mao-ming. Enhancement in detection operator extended from traditional image edge to morphology edge[J]. Journal of University of Electronic Science and Technology of China, 2005, 34(2): 206-209.
[12] 林 卉, 赵长胜, 舒 宁. 一种新的基于连通成分的边缘评价方法[J]. 现代测绘, 2003, 26(2): 8-11.
LIN Hui, ZHAO Chang-sheng, SHU Ning. A new edge evaluation method based on connection component[J]. Modern Surveying and Mapping, 2003, 26(2): 8-11.
[13] WANG Xun, JIN Jian-qiu. A edge detection algorithm based on improved canny operator[C]//Seventh International Conference on Intelligent Systems Designs and Applications. [S.l.]: [s.n.], 2007: 623-628.

编 辑 蒋 晓

(上接第419页)

[6] CHONG L, KUI W. Sensor localization with ring overlapping based on comparison of received signal strength indicator[C]//Proceedings of The 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS04). Florida, USA: IEEE, 2004.
[7] LAZOS L, POOVENDRAN R. Serloc: Robust localization for wireless sensor networks[J]. ACM Transactions on Sensor Networks, 2005, 1(1): 73-100.
[8] LAZOS L, POOVENDRAN R. HiRLoc: High-resolution robust localization for wireless sensor networks[J]. IEEE Journal on Selected Areas in Communications, 2006, 24(2): 233-246.
[9] VIVEKANANDAN V, WONG V. Concentric anchor beacon localization algorithm for wireless sensor networks[J]. IEEE Transactions on Vehicular Technology, 2007, 56: 2733-2744.
[10] HEINZELMAN W R, CHANDRAKASAN A, BALAKRISHNAN H. Energy-efficient communication protocol for wireless microsensor networks[C]//Proceeding of Hawaii International Conference on System Sciences. [S.l.]: IEEE, 2000: 10-19.
[11] 汪 炀. 无线传感器网络定位技术研究[D]. 合肥: 中国科学技术大学, 2007.
WANG Yang. A research on the localization technology of wireless sensor networks[D]. Hefei: University of Science and Technology of China, 2007.

编 辑 漆 蓉